

Binary fractions

Last lecture we discussed how to represent integers in binary. We next talk about binary representations of fractional numbers, that is, numbers that lie between the integers. Take a decimal number such as 22.63. We write this as:

$$22.63 = 2 * 10^1 + 2 * 10^0 + 6 * 10^{-1} + 3 * 10^{-2}$$

The “.” is called the *decimal point*.

We can use a similar representation for fractional binary numbers. For example,

$$(110.011)_{two} = 1 * 2^2 + 1 * 2^1 + 0 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3}$$

where “.” is called the *binary point*. If we convert to decimal, we get

$$4 + 2 + 0.25 + 0.125 = 6.375$$

Just as with integers, it is relatively straightforward to convert a binary number into a decimal number using the brute force method. To convert decimal into binary is more awkward if we use brute force. Fortunately there is a simple algorithm.

Take the example 22.63 above. We can convert the number to the left of the decimal point from decimal to binary, using the method from lecture 1, namely $22 = (10110)_{two}$. But how do we convert the fractional part (.63) to binary? The idea is to use the fact that multiplication by 2 in binary produces a shift to the left by one bit. (i.e. Multiplying by two in binary just adds 1 to each exponent in the sum of powers of two, and this corresponds to a shift of bits to the left.)

We convert 0.63 to binary as follows. To the left of the binary point, we represent the number (positive integer) in binary. To the right of the binary point, we represent the fractional part in decimal. To go from one line to the next, we multiply by two (and divide by two). To the left of the binary point, we shift by one bit and we fill the least significant bit with 1 or 0 depending on whether the (doubled) fractional part is greater than 1.

$$\begin{array}{rcll} & & . & 63 \\ = & & (1) & . 26 \times 2^{-1} \\ = & & (10) & . 52 \times 2^{-2} \\ = & & (101) & . 04 \times 2^{-3} \\ = & & (1010) & . 08 \times 2^{-4} \\ = & & (10100) & . 16 \times 2^{-5} \\ = & & (101000) & . 32 \times 2^{-6} \\ = & & (1010000) & . 64 \times 2^{-7} \\ = & & (10100001) & . 28 \times 2^{-8} \\ = & & (101000010) & . 56 \times 2^{-9} \\ = & & \text{etc} & . \text{etc} \end{array}$$

Notice that the number of bits on the left of the binary point is 9, corresponding to the shift by 9 bits which is implied by the exponent of 2^{-9} .

Finishing the example, ... since 22 in binary is 10110, we have

$$22.63 \approx 10110.101000010_{two}$$

We write \approx because we have rounded down, namely by chopping off any trailing bits..

What if we don't chop off the trailing bits and instead we continue the algorithm and generate bits *ad infinitum*. What happens? Interestingly, at some point you will generate a sequence of bits that repeats itself over and over. To see why, first consider a simple example:

$$\begin{aligned}
 & .05 \\
 = & (0) \cdot 1 \times 2^{-1} \\
 = & (00) \cdot 2 \times 2^{-2} \\
 = & (000) \cdot 4 \times 2^{-3} \\
 = & (0000) \cdot 8 \times 2^{-4} \\
 = & (00001) \cdot 6 \times 2^{-5} \\
 = & (000011) \cdot 2 \times 2^{-6} \\
 = & (0000110) \cdot 4 \times 2^{-7} \\
 = & (00001100) \cdot 8 \times 2^{-8} \\
 = & (000011001) \cdot 6 \times 2^{-9} \\
 = & (0000110011) \cdot 2 \times 2^{-10} \\
 = & \text{etc} \cdot \text{etc}
 \end{aligned}$$

But this pattern 0011 will keep repeated over and over as the part to the right of the binary point gets back to "2". This gives:

$$(.05)_{ten} = .0000\underline{11} \dots$$

where the underlined part repeats *ad infinitum*.

In a more general case, we have N digits to the right of the binary point. If "run the algorithm" I have described, then you will find that the number to the right of the binary/decimal point will eventually repeat itself. Why? Because there are only 10^N possible numbers of N digits. And once the algorithm hits an N digit number it has hit before, this defines a loop that will repeat over and over again. If the algorithm takes k steps to repeat, then k bits are generated. These k bits will similarly repeat *ad infinitum*.

Scientific Notation

We next turn to a number representation called "scientific notation," which you learned about in high school. You know that you can represent very large decimal numbers or very small decimal numbers by writing, for example:

$$300000000 = 3 \times 10^8 = 3.0\text{E} + 8$$

$$.00000456 = 4.56 \times 10^{-6} = 4.56\text{E} - 6$$

The form on the right ("E" notation) might be unfamiliar to you.

In *normalized* scientific notation, there is exactly one digit to the left of the decimal point and this digit is from 1 to 9 (not 0). *Note that the number 0 cannot be written in normalized scientific notation.*

We can do a similar thing with binary numbers. Examples of *normalized* numbers are shown on the right:

$$(1000.01)_{two} = 1.00001 \times 2^3$$

$$(0.111)_{two} = 1.11 \times 2^{-1}$$

A normalized binary number has the form:

$$1.\underline{\hspace{2cm}} \times 2^E$$

where the is called the *significand* (or *mantissa*) and E is the *exponent*. Such numbers are also called *floating point*. The number of bits in the significand is called the number of significant bits. Also, note that it is impossible to encode the number 0 in normalized form.

Floating point allows us to represent very large as well as very small numbers. An exponent of say 83 is a large number and an exponent of say -83 is a very small number (very near zero).

IEEE 754 floating point standard

To represent floating point numbers in a computer, we need to make some choices:

- how many bits to use for the significand?
- how many bit to use for the exponent?
- how to represent the exponent?
- how to represent the sign of the number?
- how to represent the number 0 ?

Until the late 1970's, different computer manufacturers made different choices. This meant that the same software produced different results when run on different computers. Obviously this was not desirable. A standard was eventually introduced: the IEEE¹ 754 standard consists of two representations for floating point numbers. One is called *single precision* and uses 32 bits. The other is called *double precision* and uses 64 bits. This representation is used in nearly all computers you will find today.

Single precision

Single precision represents a floating point number using 32 bits (4 bytes). We number the bits from right to left. (Bit 31 is the leftmost and bit 0 is the rightmost.)

- bit 31 (1 bit) is used for the sign (0 positive, 1 negative).
- bits 23-30 (8 bits) are used for the exponent
- bits 0-22 (23 bits) are used for the significand

Because the standard uses normalized numbers, the 1 bit to the left of the binary point in $1.\underline{\hspace{2cm}}$ does not need to be coded since it is always there. The significand only refers to what is to the right of the binary point.

The coding of the exponent is subtle. You might expect the exponent should be encoded with two's complement, since you want to allow positive exponents (large numbers) and negative exponents (small numbers). This is not the way the IEEE chose to do it, however.

The two exponent codes (00000000, 11111111) are *reserved* for special cases.

¹Institute of Electrical and Electronics Engineers, Inc. See <http://www.ieee.org>

Going the other way,... suppose you have a binary representation of a float and you wish to write it in IEEE format. Assume the absolute value of the number lies in $[2^{-126}, 2^{128})$ so that we can represent it as a normalized number. To represent the exponent, just add 127 (the bias) to the value of the exponent and then represent this as an 8-bit unsigned binary number. Representing the significand is easy (just strip off the first 23 bits to the right of the binary point).

How should you think about the discretization of the real line that is defined this way? Think of consecutive powers of 2, namely $[2^e, 2^{e+1}]$. That interval is of size 2^e . Partition that interval into 2^{23} equal sized pieces, each size being 2^{e-23} . Note that the step size between “samples” is constant within each interval $[2^e, 2^{e+1}]$ but that the step size doubles in each consecutive power of 2 interval.

An Example

How is 8.75 coded up in the IEEE single precision standard? First you show that

$$8.75 = (100011)_{two} \times 2^{-2}$$

using the techniques discussed earlier. Then, you shift the binary point to normalize the number,

$$(100011)_{two} \times 2^{-2} = (1.00011)_{two} \times 2^3$$

The significand is 0001100000000000000000000 where I have underlined the part that came from the above equation. Note that we have dropped the leftmost 1 bit. As discussed above, because we are using normalized numbers, we do not need to write down this 1 bit.

The exponent code is determined by adding 127 and representing the result as an 8-bit unsigned integer:

$$e = 3 + 127, \quad \text{thus,} \quad e = 130$$

and 130 written as an unsigned number is 10000010_{two}.

The sign bit is 0 since we have a positive number.

Thus, 8.75 is encoded with the 32 bits

$$0 \ 10000010 \ 000110000000000000000000.$$

We can write it in hexadecimal as follows:

$$(01000001000011000000000000000000)_{two} = 0x410c0000$$

Double precision

The 23 bits of significand might seem like alot at first glance. But notice that 2^{23} is only about 8 million. When we try to represent integers larger than this value, we find that we cannot do so. (See Exercises 1.)

In order to get more precision, we need more bits. The IEEE double precision representation uses 64 bits instead of 32. 1 bit is used for the sign, 11 bits for the exponent, and remaining 52 bits for the significand ($1 + 11 + 52 = 64$). A similar idea is used for the exponent bias except that now the bias is 1023 ($2^{10} - 1$), rather than 127 ($2^7 - 1$). Conceptually, there is nothing new here beyond single precision, though.

Further stuff on slides

- I discussed how powers of 2 can be related to powers of 10 by noting that $2^{10} = 1024 \approx 10^3$. For example, 23 bits of precision corresponds roughly to 7 decimal digits of precision e.g. $2^{23} \approx 10^7$. (In class, I argued that $2^{-23} \approx 10^{-7}$ but the idea is the same.) Similarly, 52 bits of precision for doubles corresponds roughly to about 16 decimal digits.
- I gave an example of some simple Java code that demonstrates the limited precision of both single and double precision numbers.
- I noted that laws such as the associativity of addition, namely $(a + b) + c = a + (b + c)$, don't necessarily hold when you are dealing with floating point numbers. Avoiding such problems is important (and fortunately, it is possible, as you will learn if you take COMP 350).